PARALLEL PROGRAMMING ON MULTICORES

Threading Subroutines

Bo Zhang
zhangb@cs.duke.edu

Nikos P. Pitsianis
Xiaobai Sun

DEPARTMENT OF COMPUTER SCIENCE
Duke University

09/29/2010
**My Experience**

**Background**
Ph.D. in Mathematics; basic knowledge of C

**What I have gained**

- Learned Fortran, Pthread, and OpenMP
- Parallel Fast Multipole Method (FMM)

1. 1st parallel version on multicore desktop & laptop
2. Initial code has 12k lines; current code has 18k lines
3. In both Pthread and OpenMP
4. Problem size: 20M (laptop); 100M (workstation)
5. Speedup: 8X on an oct-core workstation
You have a working sequential code, but the parallel version sees

1. Segmentation fault
2. Bus error
3. Heap collapse
4. Inconsistent results

What happened?
USE OF EXISTING SUBROUTINES

Potential Problems, not \textsc{ThreadSafe}

- Concurrent updates to global variables
- Concurrent use of local scratch space
- Stack overflow
Concurrent updates to global variables

```c
double balance;
void deposit (double s) {
    balance += s;
}
void withdraw (double s) {
    balance -= s;
}
int main () {
    balance = 0.0
    . . .
    printf("%f\n",balance);
    return 0;
}
```
Use of Existing Subroutines

Concurrent updates to global variables

```c
double balance;
void deposit (double s) {
    balance += s;
}
void withdraw (double s) {
    balance -= s;
}
int main () {
    balance = 0.0
    deposit (s1);
    deposit (s2);
    withdraw (s3);
    ...
    printf("%f\n",balance);
    return 0;
}
```

💡 Use mutex variable
Concurrent updates to global variables

```c
double balance;
void deposit (double s) {
    balance += s;
}
void withdraw (double s) {
    balance -= s;
}
int main () {
    balance = 0.0
    deposit (s1);
    deposit (s2);
    withdraw (s3);
    ...
    printf("%f\n",balance);
    return 0;
}
```

Use mutex variable

```c
double balance;
pthread_mutex_t balance_mutex;
void deposit (double s) {
    pthread_mutex_lock (&balance_mutex);
    balance += s;
    pthread_mutex_unlock (&balance_mutex);
}
void withdraw (double s) {
    pthread_mutex_lock (&balance_mutex);
    balance -= s;
    pthread_mutex_unlock (&balance_mutex);
}
int main () {
    balance = 0.0;
    ...
    printf("%f\n",balance);
    return 0;
}
```
Use of Existing Subroutines

Concurrent use of local scratch space

```c
int main () {
    int s1, s2, s, work[100];
    s1 = task1(work);
    s2 = task2(work);
    s = s1+s2;
    printf("%d\n", s);
    return 0;
}
```
Concurrent use of local scratch space

```c
int main () {
    int s1, s2, s, work[100];
    s1 = task1(work);
    s2 = task2(work);
    s = s1+s2;
    printf("%d\n", s);
    return 0;
}
```

Allocate for each thread; trim it down when necessary
Concurrent use of local scratch space

Allocate for each thread; trim it down when necessary
USE OF EXISTING SUBROUTINES

Stack overflow

Caused by recursive call of subroutine

```c
void foo (int i, int *s) {
    int ia[100];
    if (i > 0) {
        foo(i-1, s);
        *s += compute(ia, i);
    }
}

int main () {
    int s, i = 2;
    foo(i, &s);
    return 0;
}
```
USE OF EXISTING SUBROUTINES

Stack overflow

Caused by recursive call of subroutine

```c
void foo (int i, int *s) {
    int ia[100];
    if ( i > 0 )
        foo(i-1, s);
    *s += compute(ia, i);
}

int main () {
    int s, i = 2;
    foo(i, &s);
    return 0;
}
```
USE OF EXISTING SUBROUTINES

Stack overflow

Caused by recursive call of subroutine

void foo (int i, int *s) {
    int ia[100];
    if (i > 0)
        foo(i-1, s);
    *s += compute(ia, i);
}

int main () {
    int s, i = 2;
    foo(i, &s);
    return 0;
}
**Use of Existing Subroutines**

Stack overflow

Caused by recursive call of subroutine

```c
void foo (int i, int *s) {
    int ia[100];
    if ( i > 0 )
        foo(i-1, s);
    *s += compute(ia, i);
}

int main () {
    int s, i = 2;
    foo(i, &s);
    return 0;
}
```
USE OF EXISTING SUBROUTINES

Stack overflow

Caused by recursive call of subroutine

```
void foo (int i, int *s) {
    int ia[100];
    if ( i > 0)
        foo(i-1, s);
    *s += compute(ia, i);
}

int main () {
    int s, i = 2;
    foo(i, &s);
    return 0;
}
```
USE OF EXISTING SUBROUTINES

Stack overflow

Caused by recursive call of subroutine

Unfold recursion

void foo(int i, int *s) {
    int ia[100];
    if ( i > 0 )
        foo(i-1, s);
    *s += compute(ia, i);
}

int main () {
    int i, s, ia[100];
    for ( i = 0; i < 2; i++ )
        s += compute(ia,i);
    return 0;
}
USE OF EXISTING SUBROUTINES

Stack overflow

Caused by memory allocation within subroutine

```c
int ps[3];

void *foo (void *threadid) {
    long tid = (long)threadid;
    int ia[100];
    ps[tid] = compute(ia,i);
    pthread_exit(NULL);
}

int main () {
    long i;
    int s;
    pthread_t threads[3];
    for ( i = 0; i < 3; i++ )
        pthread_create(&threads[i], NULL, foo, (void *)i);
    for ( i = 0; i < 3; i++ )
        pthread_join (threads[i], NULL);
    s = ps[0]+ps[1]+ps[2];
    return 0;
}
```
USE OF EXISTING SUBROUTINES

Stack overflow

Caused by memory allocation within subroutine

- Dynamic allocation with `malloc()`
- Use pthread library `pthread_attr_setstacksize()`
- Allocate within `main()`, pass pointer to thread
USE OF EXISTING SUBROUTINES

Stack overflow

Caused by memory allocation within subroutine

Allocate within `main()`, pass pointer to thread

```c
int ps[3];
int *gia;

void *foo (void *threadid) {
    long tid = (long)threadid;
    int *ia = &gia[tid*100];
    ps[tid] = compute(ia,i);
    pthread_exit(NULL);
}

int main () {
    long i;
    int s;
    pthread_t threads[3];
    gia = (int *)malloc(sizeof(int)*3*100);
    for ( i = 0; i < 3; i++ )
        pthread_create(&threads[i], NULL, foo, (void *)i);
    for ( i = 0; i < 3; i++ )
        pthread_join (threads[i], NULL);
    s = ps[0]+ps[1]+ps[2];
    free(gia);
    return 0;
}
```
Algorithmic Reduction of Critical Sections

Reading & Writing are not symmetric

1. Concurrent reading is fast & safe
2. Concurrent writing must be AVOIDED
Algorithmic Reduction of Critical Sections

Example: $T_{ML}$ of FMM
Algorithmic Reduction of Critical Sections

Example: $T_{ML}$ of FMM

\begin{figure}[h]
\centering
\includegraphics[width=0.8\textwidth]{example_diagram.png}
\caption{Graphical representation of $T_{ML}$ of FMM}
\end{figure}
Algorithmic Reduction of Critical Sections

Example: $T_{ML}$ of FMM

\[ \begin{array}{cccc}
\cdots & \cdots & \cdots & \cdots \\
\cdots & \cdots & \cdots & \cdots \\
\cdots & \cdots & \cdots & \cdots \\
\cdots & \cdots & \cdots & \cdots \\
\end{array} \]

Need $O(N)$ Locks!!!

\[ \begin{array}{cccc}
\cdots & \cdots & \cdots & \cdots \\
\cdots & \cdots & \cdots & \cdots \\
\cdots & \cdots & \cdots & \cdots \\
\cdots & \cdots & \cdots & \cdots \\
\end{array} \]

Need $O(3d - 2d)$ Locks

Need ZERO Locks
Algorithmic Reduction of Critical Sections

Example: $\mathcal{T}_{ML}$ of FMM
Algorithmic Reduction of Critical Sections

Example: $T_{ML}$ of FMM
Algorithmic Reduction of Critical Sections

Example: $\mathcal{T}_{ML}$ of FMM

Need $O(N)$ Locks!!!
ALGORITHMIC REDUCTION OF CRITICAL SECTIONS

Example: $T_{ML}$ of FMM

Need $O(3^d - 2^d)$ Locks
Algorithmic Reduction of Critical Sections

Example: $T_{ML}$ of FMM

Need ZERO Locks
Developing Tips

💡 Save a copy of your working code
💡 Do sequential version first
💡 Work on one subroutine at a time
💡 Pick the most time-critical subroutine first
Developing Tips

Study software structures

Time Step

Upward Pass: $T_{SM}, T_{MM}$

List 2: $T_{ME}, T_{EE}, T_{EL}$

List 3: $T_{MT} \text{ or } T_{ST}$

List 5 & Evaluate Local: $T_{LL} \text{ & } T_{LT}$

List 4: $T_{SL} \text{ or } T_{ST}$

List 1: $T_{ST}$

Operations

All $l$'s

All $l$'s

Sum Local & Direct

All Particles
Do **NOT** use unless necessary

- No `goto` statement in Fortran
- Loops in Fortran
- No `break` statement

**Utilize Compiler instead of blocking it**
Coding Tips

Do **NOT** use unless necessary

- **No goto statement in Fortran**
- **Loops in Fortran**

```f90
DO ...... 
       ...... 
       ENDDO
```

```f90
DO 10 ...... 
       ...... 
10 CONTINUE
```

- **No break statement**

Utilize Compiler instead of blocking it
Do **NOT** use unless necessary

**No goto statement in Fortran**

**Loops in Fortran**

**No break statement**

```c
while (cond) {
    ....
    cond = foo();
}
```

```c
while (1) {
    ....
    break;
}
```

**Utilize Compiler instead of blocking it**
Testing & Tuning Tips

- Makefile
- Split Fortran subroutines into individual file
- Debugging & profiling tools
- Shell script for batch test
Testing & Tuning Tips

- **Makefile**
  - Split Fortran subroutines into individual file
    - Fsplt
  - Use individual compilation flag for each routine

- **Debugging & profiling tools**
- **Shell script for batch test**
TESTING & TUNING TIPS

💡 Makefile

💡 Split Fortran subroutines into individual file

💡 Debugging & profiling tools

  √ gprof
  http://www.cs.utah.edu/dept/old/texinfo/as/gprof_toc.html

  √ DDD
  http://www.gnu.org/software/ddd

  √ valgrind
  http://valgrind.org/

💡 Shell script for batch test
TESTING & TUNING TIPS

Makefile

Split Fortran subroutines into individual file

Debugging & profiling tools

Shell script for batch test

```bash
#!/bin/bash
for flag in -O -O2 -O4
do
    make clean
    make CFLAGS=$flag fFLAGS=$flag sFLAGS=-O
    for (( t=2; t<=128; t=t*2 ))
do
        for (( s=50; s<=90; s=s+5 ))
do
            for (( count=1; count<=10; count++ ))
do
                ./fmm -n 10000 -t $t -s $s -d 2
done
done
done

> nohup ./script &
```
Summary

- Threading subroutines
- Algorithmic reduction of critical sections
- Programming development tips

Thank you for your attention
Please send your comments to zhangb@cs.duke.edu